Java 8 : Stream

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Plan

Plan

8. Quantificateurs `...Match()`
   - `anyMatch()`
   - `allMatch()`
   - `noneMatch()`
9. Fusion de streams
   - `flatMap`
   - `distinct`
10. `IntStream`, `LongStream` et `DoubleStream`

Java

Quelques nouvelles méthodes pour les collections

- Plusieurs méthodes ont été ajoutées dans la hiérarchie des collections
- La plupart de ces méthodes utilisent les expressions lambda
- Objectif : éviter les itérations, simplifier l'écriture de code et améliorer la lisibilité

Java

Considérons la liste suivante

```
List<Integer> liste = new ArrayList<>(Arrays.asList(2, 7, 1, 3));
```

Java

Considérons la liste suivante

```
List<Integer> liste = new ArrayList<>(Arrays.asList(2, 7, 1, 3));
```

Pour parcourir une liste, on peut utiliser la méthode `forEach` qui prend en paramètre une expression lambda de type `Consumer`

```
liste.forEach(elt -> System.out.println(elt));
```

Java

Considérons la liste suivante

```
List<Integer> liste = new ArrayList<>(Arrays.asList(2, 7, 1, 3));
```

Pour parcourir une liste, on peut utiliser la méthode `forEach` qui prend en paramètre une expression lambda de type `Consumer`

```
liste.forEach(elt -> System.out.println(elt));
```

Résultat

```
2
7
1
3
```

Java

Ou en plus simple avec les références de méthode

```
liste.forEach(System.out::println);
```

Java

Ou en plus simple avec les références de méthode

```
liste.forEach(System.out::println);
```

Résultat

```
2
7
1
3
```

Java

Pour supprimer selon une condition, on utilise la méthode `removeIf()` qui prend en paramètre une expression lambda de type `Predicate`

```
liste.removeIf(elt -> elt > 6);
liste.forEach(System.out::println);
```

Java

Pour supprimer selon une condition, on utilise la méthode `removeIf()` qui prend en paramètre une expression lambda de type `Predicate`

```
liste.removeIf(elt -> elt > 6);
liste.forEach(System.out::println);
```


Résultat

```
2
1
3
```

Java

Pour modifier tous les élément de la liste, on utilise la méthode `replaceAll()` qui prend en paramètre une expression lambda de type `UnaryOperator`

```
liste.replaceAll(elt -> elt + 6);
liste.forEach(System.out::println);
```

Java

Pour modifier tous les élément de la liste, on utilise la méthode `replaceAll()` qui prend en paramètre une expression lambda de type `UnaryOperator`

```
liste.replaceAll(elt -> elt + 6);
liste.forEach(System.out::println);
```


Résultat

```
8
13
7
9
```

Java

Pour parcourir un `Map` avec `forEach`

```
Map<Integer, String> fcb = new HashMap<>();
fcb.put(10, "Messi");
fcb.put(9, "Suarez");
fcb.put(23, "Umtiti");
fcb.put(4, "Rakitic");
fcb.forEach((k, v) -> System.out.println(k + " " + v));
```

Java

Pour parcourir un `Map` avec `forEach`

```
Map<Integer, String> fcb = new HashMap<>();
fcb.put(10, "Messi");
fcb.put(9, "Suarez");
fcb.put(23, "Umtiti");
fcb.put(4, "Rakitic");
fcb.forEach((k, v) -> System.out.println(k + " " + v));
```


Résultat

```
4 Rakitic
23 Umtiti
9 Suarez
10 Messi
```

Java

Pour appliquer un traiteme nt si la clé est présente

```
fcb.computeIfPresent(10, (k, v) -> v.toUpperCase());
fcb.forEach((k, v) -> System.out.println(k + " " + v));
```

Java

Pour appliquer un traiteme nt si la clé est présente

```
fcb.computeIfPresent(10, (k, v) -> v.toUpperCase());
fcb.forEach((k, v) -> System.out.println(k + " " + v));
```

Résultat

```
4 Rakitic
23 Umtiti
9 Suarez
10 MESSI
```

Java

L'API **Stream** (disponible depuis **Java 8**)

- **Stream** : traduit en flux (de données)
- Nouvelle API pour la manipulation de données
- Simplifiant la recherche, le filtrage et la manipulation de données...
- Ne modifiant pas forcément la source de données

Java

L'API **Stream** (disponible depuis **Java 8**)

- **Stream** : traduit en flux (de données)
- Nouvelle API pour la manipulation de données
- Simplifiant la recherche, le filtrage et la manipulation de données...
- Ne modifiant pas forcément la source de données

Les méthodes de Stream se trouvent dans

```
java.util.stream;
```

Java

Un stream

- un chaînage d'opération
- 0 ou plusieurs opérations intermédiaires
  - retourne toujours un stream
  - n'est exécuté qu'en appelant l'opération terminale
- 1 opération terminale
  - consomme le stream
  - exécute les opérations intermédiaires

Java

Exemples d'opération intermédiaire

- `map(Function)` : permet d'effectuer un traitement sur une liste sans la modifier réellement
- `filter(Predicate)` : permet de filtrer des éléments selon un prédicat
- `...`

Java

Exemples d'opération finale

- `forEach(Consumer)`
- `reduce()` : permet de réduire une liste en une seule valeur
- `count()` : permet de compter le nombre d'élément d'une liste
- `collect()` : permet de récupérer le résultat des opérations successives sous une certaine forme à spécifier
- `...`

Java

Un Stream peut être construit à partir de

- une suite de données
- un tableau
- une liste

Java

Pour construire un Stream à partir d'une suite de données

```
Stream stream = Stream.of(2, 7, 1, 3);
```

Java

Pour construire un Stream à partir d'une suite de données

```
Stream stream = Stream.of(2, 7, 1, 3);
```

On peut utiliser la méthode `of()` pour construire un Stream à partir d'un tableau

```
int[] tab = { 2, 7, 1, 3 };
Stream stream = Stream.of(tab);
```

Java

On peut utiliser la méthode `stream()` pour construire un Stream à partir d'une liste

```
List<Integer> liste = Arrays.asList(2, 7, 1, 3);
Stream stream =  liste.stream();
```

Java

Si les données sont de même type, on peut utiliser la généricité

```
List<Integer> liste = Arrays.asList(2, 7, 1, 3);
Stream<Integer> stream =  liste.stream();
```

Java

Pour parcourir et afficher une liste

```
stream.forEach(elt -> System.out.println(elt));
```

Java

Pour parcourir et afficher une liste

```
stream.forEach(elt -> System.out.println(elt));
```

Résultat

```
2
7
1
3
```

Java

Une deuxième écriture possible de `forEach` avec les références de méthode

```
stream.forEach(System.out::println);
```

Java

Une deuxième écriture possible de `forEach` avec les références de méthode

```
stream.forEach(System.out::println);
```

Résultat

```
2
7
1
3
```

Java

Si l'ordre n'a pas d'importance, on peut utiliser `parallelStream` qui permet une exécution parallèle du programme et donc une meilleure performance

```
Stream<Integer> stream = liste.parallelStream();
stream.forEach(System.out::println);
```

Java

Si l'ordre n'a pas d'importance, on peut utiliser `parallelStream` qui permet une exécution parallèle du programme et donc une meilleure performance

```
Stream<Integer> stream = liste.parallelStream();
stream.forEach(System.out::println);
```

Résultat

```
1
7
2
3
```

Java

Stream séquentiel

- Utilise un seule core.
- Une seule itération à un instant `t`.
- Moins Performant.
- Ordre maintenu.
- Plus fiable et moins d'erreur..

Stream parallèle

- Utilise plusieurs cores.
- Autant d'itérations que de cores disponibles à un instant `t`.
- Plus performant.
- Ordre non maintenu.
- Moins fiable et risque d'erreur plus élevé.

Java

Exemple avec une opération intermédiaire `map` et une opération finale `forEach`

```
stream
        .map(elt -> elt + 2)
        .forEach(elt -> System.out.println(elt));
```

Java

Exemple avec une opération intermédiaire `map` et une opération finale `forEach`

```
stream
        .map(elt -> elt + 2)
        .forEach(elt -> System.out.println(elt));
```

Résultat

```
4
9
3
5
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```


Exercice

En utilisant les **Stream**, écrire un code qui permet d'afficher la longueur de chaque marque du tableau `marques`.

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```


Exercice

En utilisant les **Stream**, écrire un code qui permet d'afficher la longueur de chaque marque du tableau `marques`.

Résultat attendu

```
7
4
8
6
```

Java

Exemple avec deux opérations intermédiaires `map` et `filter`

```
stream
        .map(elt -> elt + 2)
        .filter(elt -> elt > 3)
        .forEach(elt -> System.out.println(elt));
```

Java

Exemple avec deux opérations intermédiaires `map` et `filter`

```
stream
        .map(elt -> elt + 2)
        .filter(elt -> elt > 3)
        .forEach(elt -> System.out.println(elt));
```


Résultat

```
4
9
5
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```


Exercice 1

En utilisant les **Stream**, écrire un code qui permet d'afficher les `marques` contenant un nombre impair de caractères.

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList("peugeot", "ford", "mercedes", "
   cooper");
```


Exercice 1

En utilisant les **Stream**, écrire un code qui permet d'afficher les `marques` contenant un nombre impair de caractères.

Résultat attendu

```
peugeot
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

En utilisant les **Stream**, écrire un code qui permet d'afficher les noms de personnes majeures.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

En utilisant les **Stream**, écrire un code qui permet d'afficher les noms de personnes majeures.

Résultat attendu

```
dalton
benamar
```

Java

L'opération `reduce`

- permet de réduire le stream en une seule valeur (opération finale)
- retourne le résultat sous forme d'un `Optional`
- s'exprime avec une expression lambda avec deux paramètres d'entrée :
  - le premier correspond à la valeur de retour de la l'application précédente et
  - le deuxième contient l'élément courant

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b);
if (somme.isPresent())
  System.out.println(somme.get());
// affiche 18
```

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b);
if (somme.isPresent())
  System.out.println(somme.get());
// affiche 18
```

Pour éviter le test, on peut faire

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b);
somme.ifPresent(System.out::print);
// affiche 18
```

La méthode `reduce` retourne la somme si la liste filtrée n'est pas vide sinon elle ne retourne rien. Il faut donc tester la présence d'un résultat avant de l'afficher

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b);
if (somme.isPresent())
  System.out.println(somme.get());
// affiche 18
```

Pour éviter le test, on peut faire

```
Optional<Integer> somme = stream.map(elt -> elt + 2)
                .filter(elt -> elt > 3)
                .reduce((a, b) -> a + b);
somme.ifPresent(System.out::print);
// affiche 18
```

Ou aussi

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3)
      .reduce((a, b) -> a + b)
      .ifPresent(System.out::print);
// affiche 18
```

Java

Pour éviter de retourner un `Optional`, on peut initialiser le premier paramètre `a` à 0.

```
int somme = stream.map(elt -> elt + 2)
          .filter(elt -> elt > 3)
          .reduce(0, (a, b) -> a + b);
System.out.println(somme);
// affiche 18
```

Java

Pour éviter de retourner un `Optional`, on peut initialiser le premier paramètre `a` à 0.

```
int somme = stream.map(elt -> elt + 2)
          .filter(elt -> elt > 3)
          .reduce(0, (a, b) -> a + b);
System.out.println(somme);
// affiche 18
```

Ou aussi

```
int somme = stream.map(elt -> elt + 2)
          .filter(elt -> elt > 3)
          .reduce((a, b) -> a + b)
          .orElse(0);
System.out.println(somme);
// affiche 18
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList(
        "peugeot", "ford", "mercedes", "cooper"
);
```

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList(
        "peugeot", "ford", "mercedes", "cooper"
);
```


Exercice 1

En utilisant les **Stream**, écrire un code qui permet de calculer le nombre total de caractères de toutes les chaînes du tableau `marques`.

Java

Considérons la liste suivante

```
List<String> marques = Arrays.asList(
        "peugeot", "ford", "mercedes", "cooper"
);
```


Exercice 1

En utilisant les **Stream**, écrire un code qui permet de calculer le nombre total de caractères de toutes les chaînes du tableau `marques`.

Résultat attendu

```
25
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

En utilisant les **Stream**, écrire un code qui permet de calculer la moyenne d'age de personnes majeures.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice 2

En utilisant les **Stream**, écrire un code qui permet de calculer la moyenne d'age de personnes majeures.

Résultat attendu

```
19
```

Java

findFirst

- permet de réduire le stream en une seule valeur (opération finale)
- retourne le premier élément de la sélection sous forme d'un `Optional`

Java

Pour sélectionner le premier élément de la liste qui commence par 'A'

```
List<String> liste = Arrays.asList(
        "Jack", "Ahmed", "John", "Alain", "Jasmine"
);

Optional<String> findFirst = liste.stream()
        .filter(s -> s.startsWith("A"))
        .findFirst();

System.out.println(findFirst.get());
// affiche Ahmed
```

Java

findAny

- permet de réduire le stream en une seule valeur (opération finale)
- retourne un élément quelconque de la sélection sous forme d'un `Optional`

Java

Pour sélectionner un élément de la liste qui commence par 'A'

```
List<String> liste = Arrays.asList(
        "Jack", "Ahmed", "John", "Alain", "Jasmine"
);

Optional<String> findFirst = liste.stream()
        .filter(s -> s.startsWith("A"))
        .findAny();

System.out.println(findFirst.get());
// affiche Ahmed
```

Java

Remarques

- À première vue, `findAny` et `findAny` retournent le même résultat.
- Dans un stream parallèle, `findAny` ne retourne pas forcément le premier élément qui remplit la condition.

Java

Exécutez plusieurs fois le code suivant vérifiez que la valeur sélectionnée n'est pas toujours la même

```
List<String> liste = Arrays.asList(
        "Jack", "Ahmed", "John", "Alain", "Jasmine"
);

Optional<String> findFirst = liste.parallelStream()
        .filter(s -> s.startsWith("A"))
        .findAny();

System.out.println(findFirst.get());
// affiche Ahmed ou Alain
```

Java

Appliquons les modifications suivante à la liste précédente

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Java

Appliquons les modifications suivante à la liste précédente

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Affichons ensuite les éléments de la liste

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Java

Appliquons les modifications suivante à la liste précédente

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```

Affichons ensuite les éléments de la liste

```
stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3);
liste.forEach(elt -> System.out.println(elt));
```


Résultat

```
2
7
1
3
```

Java

Pour enregistrer les modifications de `map` et `filter`

```
liste = stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3)
      .collect(Collectors.toList());
liste.forEach(elt -> System.out.println(elt));
```

Java

Pour enregistrer les modifications de `map` et `filter`

```
liste = stream.map(elt -> elt + 2)
      .filter(elt -> elt > 3)
      .collect(Collectors.toList());
liste.forEach(elt -> System.out.println(elt));
```


Résultat

```
4
9
5
```

Java

Remarque

Il est possible d'obtenir un résultat sous forme

- d'un `Set` avec la méthode `toSet()`
- d'un `map()` avec la méthode `toMap()`
- ...

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de récupérer, dans une liste, les noms de personnes présentes dans la liste `personnes`.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de récupérer, dans une liste, les noms de personnes présentes dans la liste `personnes`.

Résultat attendu

```
System.out.println(noms);
// affiche [wick, dalton, maggio, benamar]
```

Java

Pour transformer la liste en Map

```
liste.stream()
.collect(Collectors.toMap(
    elt -> elt,
    elt -> elt * 2
))
.forEach((k, v)-> System.out.println(k + " " + v));
```

Java

Pour transformer la liste en Map

```
liste.stream()
.collect(Collectors.toMap(
    elt -> elt,
    elt -> elt * 2
))
.forEach((k, v)-> System.out.println(k + " " + v));
```


Résultat

```
1 2
2 4
3 6
7 14
```

Java

Exercice

En utilisant les **Stream**, écrire un code qui permet de construire un `Map` à partir de la liste `personnes`, les clés seront les noms et les valeurs seront les ages.

Java

Exercice

En utilisant les **Stream**, écrire un code qui permet de construire un `Map` à partir de la liste `personnes`, les clés seront les noms et les valeurs seront les ages.

Résultat attendu

```
dalton 18
wick 16
maggio 17
benamar 20
```

Java

On peut aussi transformer la liste en Map en utilisant `groupingBy` : la clé est la première lettre du prénom, la valeur est la liste des personnes dont le prénom commence par cette lettre

```
personnes.stream()
    .collect(Collectors.groupingBy(elt -> elt.getPrenom().charAt(0)))
    .forEach((k, v)-> System.out.println(k + " " + v));
```

Java

On peut aussi transformer la liste en Map en utilisant `groupingBy` : la clé est la première lettre du prénom, la valeur est la liste des personnes dont le prénom commence par cette lettre

```
personnes.stream()
     .collect(Collectors.groupingBy(elt -> elt.getPrenom().charAt(0)))
     .forEach((k, v)-> System.out.println(k + " " + v));
```

Résultat

```
s [Personne [nom=benamar, prenom=sophie, age=20]]
c [Personne [nom=maggio, prenom=carol, age=17]]
j [Personne [nom=wick, prenom=john, age=16], Personne [nom=dalton,
   prenom=jack, age=18]]
```

Java

Pour calculer la moyenne

```
Double moyenne = stream.collect(Collectors.
   averagingInt(Integer::intValue));

System.out.println(moyenne);
// affiche 3.25
```

Java

Pour calculer la somme

```
Integer somme = stream.collect(Collectors.summingInt
   (Integer::intValue));

System.out.println(somme);
// affiche 13
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner la moyenne d'age de personnes majeures.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner la moyenne d'age de personnes majeures.

Résultat attendu

```
19
```

Java

On peut aussi compter le nombre d'éléments

```
long nbr = stream.map(elt -> elt + 2)
      .filter(elt -> elt > 5)
      .count();
System.out.println(nbr);
// affiche 1
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de compter le nombre de personnes majeures.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de compter le nombre de personnes majeures.

Résultat attendu

```
2
```

Java

On peut aussi compter le nombre d'éléments

```
long nbr = stream.map(elt -> elt + 2)
      .filter(elt -> elt > 5)
      .count();
System.out.println(nbr);
// affiche 1
```

Java

On peut aussi compter le nombre d'éléments

```
long nbr = stream.map(elt -> elt + 2)
      .filter(elt -> elt > 5)
      .count();
System.out.println(nbr);
// affiche 1
```

Pour chercher le min ou le max

```
int nbr = stream
      .max(Comparator.naturalOrder()).get();
System.out.println(nbr);
// affiche 7
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```


Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner le plus jeune age.

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner le plus jeune age.

Résultat attendu

```
16
```

Java

Solution

```
personnes.stream()
     .map(Personne::getAge)
     .min(Comparator.naturalOrder())
     .ifPresent(System.out::println);
```

Java

Solution

```
personnes.stream()
     .map(Personne::getAge)
     .min(Comparator.naturalOrder())
     .ifPresent(System.out::println);
```

Pour récupérer la personne la plus jeune de la liste

```
personnes.stream()
     .min(Comparator.comparing(Personne::getAge))
     .ifPresent(System.out::println);
```

Java

Solution

```
personnes.stream()
     .map(Personne::getAge)
     .min(Comparator.naturalOrder())
     .ifPresent(System.out::println);
```

Pour récupérer la personne la plus jeune de la liste

```
personnes.stream()
     .min(Comparator.comparing(Personne::getAge))
     .ifPresent(System.out::println);
```

Ou en utilisant `collect`

```
personnes.stream()
     .collect(Collectors.minBy(Comparator.comparing(Personne::getAge)))
     .ifPresent(System.out::println);
```

Java

Pour limiter le nombre d'éléments sélectionnés

```
stream
      .limit(3)
      .forEach(System.out::println);
```

Java

Pour limiter le nombre d'éléments sélectionnés

```
stream
      .limit(3)
      .forEach(System.out::println);
```

Résultat

```
2
7
1
```

Java

Pour skipper quelques éléments

```
stream
      .skip(2)
      .forEach(System.out::println);
```

Java

Pour skipper quelques éléments

```
stream
      .skip(2)
      .forEach(System.out::println);
```


Résultat

```
1
3
```

Java

Pour trier les éléments

```
stream
      .sorted()
      .forEach(System.out::println);
```

Java

Pour trier les éléments

```
stream
      .sorted()
      .forEach(System.out::println);
```

Résultat

```
1
2
3
7
```

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```


Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner les trois premières personnes les plus jeunes dans l'ordre.

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(Arrays.asList(
    new Personne("wick", "john", 16),
    new Personne("dalton", "jack", 18),
    new Personne("maggio", "carol", 17),
    new Personne("benamar", "sophie", 20)
));
```


Exercice

En utilisant les **Stream**, écrire un code qui permet de retourner les trois premières personnes les plus jeunes dans l'ordre.

Résultat attendu

```
Personne[nom=wick, prenom=john, age=16]
Personne[nom=maggio, prenom=carol, age=17]
Personne[nom=dalton, prenom=jack, age=18]
```

Java

Pour tester s'il existe un élément qui vérifie une condition, on peut utiliser `anyMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .anyMatch(element -> element == 9);

System.out.println(result);
// affiche true
```

Java

Pour tester si tous les éléments vérifient une condition, on peut utiliser `allMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .allMatch(element -> element == 9);

System.out.println(result);
// affiche false
```

Java

Pour tester aucun élément ne vérifie une condition, on peut utiliser `noneMatch(Predicate)` qui retourne un booléen

```
boolean result = stream
    .map(elt -> elt + 2)
    .noneMatch(element -> element >= 15);

System.out.println(result);
// affiche true
```

Java

Ajoutons une liste de notes comme attribut dans la classe `Personne`

```
package org.eclipse.model;

import java.util.List;

public class Personne {
        private String nom;
        private String prenom;
        private int age;
        private List<Integer> notes;

        // N'oublions pas d'ajouter un constructeur
           à 4 paramètres, de modifier le toString
           et de générer les getters et setters
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(
    Arrays.asList(
        new Personne("wick", "john", 16, List.of(10, 12, 15)),
        new Personne("dalton", "jack", 18, List.of(8, 18, 5)),
        new Personne("maggio", "carol", 17, List.of(18, 15, 14, 13)),
        new Personne("benamar", "sophie", 20,  List.of(8, 8, 18))
    )
);
```

Java

Considérons la liste suivante

```
List<Personne> personnes = new ArrayList<>(
    Arrays.asList(
        new Personne("wick", "john", 16, List.of(10, 12, 15)),
        new Personne("dalton", "jack", 18, List.of(8, 18, 5)),
        new Personne("maggio", "carol", 17, List.of(18, 15, 14, 13)),
        new Personne("benamar", "sophie", 20,  List.of(8, 8, 18))
    )
);
```


Exercice

En utilisant les **Stream**, écrire un code qui permet d'afficher toutes les notes.

Java

Première solution

```
personnes.stream()
     .map(Personne::getNotes)
     .forEach(System.out::println);
```

Java

Première solution

```
personnes.stream()
     .map(Personne::getNotes)
     .forEach(System.out::println);
```

Résultat

```
[10, 12, 15]
[8, 18, 5]
[18, 15, 14, 13]
[8, 8, 18]
```

Java

Constat

Le stream contient 4 sous-listes de note.

Java

Constat

Le stream contient 4 sous-listes de note.

Question

Et si on voulait que tous les éléments soit dans la même liste ?

Java

Constat

Le stream contient 4 sous-listes de note.

Question

Et si on voulait que tous les éléments soit dans la même liste ?

Solution

Utiliser `flapMap`.

Java

Deuxième solution avec `flatMap`

```
personnes.stream()
     .flatMap(elt -> elt.getNotes().stream())
     .forEach(System.out::println);
```

Java

Deuxième solution avec `flatMap`

```
personnes.stream()
     .flatMap(elt -> elt.getNotes().stream())
     .forEach(System.out::println);
```

Résultat

```
10
12
15
8
18
5
18
15
14
13
8
8
18
```

Java

Pour supprimer les doublons, on utilise `distinct`

```
personnes.stream()
     .flatMap(elt -> elt.getNotes().stream())
     .distinct()
     .forEach(System.out::println);
```

Java

Pour supprimer les doublons, on utilise `distinct`

```
personnes.stream()
     .flatMap(elt -> elt.getNotes().stream())
     .distinct()
     .forEach(System.out::println);
```

Résultat

```
10
12
15
8
18
5
14
13
```

Java

Remarque

Si toutes les données d'un Stream sont des nombres, on peut utiliser des Stream particuliers :

- `IntStream`
- `DoubleStream`
- `LongStream`

Java

Si toutes les valeurs sont de type entier, on peut utiliser `IntStream`

```
IntStream stream = IntStream.of(1, 2, 3);
stream.forEach(System.out::println);
```

Java

Si toutes les valeurs sont de type entier, on peut utiliser `IntStream`

```
IntStream stream = IntStream.of(1, 2, 3);
stream.forEach(System.out::println);
```

Résultat

```
1
2
3
```

Java

On peut aussi obtenir un `IntStream` à partir d'un tableau prédéfini

```
int [] tab = { 1, 2, 3 };
IntStream stream = Arrays.stream(tab);
stream.forEach(System.out::println);
```

Java

On peut aussi obtenir un `IntStream` à partir d'un tableau prédéfini

```
int [] tab = { 1, 2, 3 };
IntStream stream = Arrays.stream(tab);
stream.forEach(System.out::println);
```


Résultat

```
1
2
3
```

Java

On peut aussi obtenir un `IntStream` à partir d'une liste prédéfinie

```
List<Integer> liste = Arrays.asList(1, 2, 3);
IntStream stream = liste.stream().mapToInt(Integer::intValue);
stream.forEach(System.out::println);
```

Java

On peut aussi obtenir un `IntStream` à partir d'une liste prédéfinie

```
List<Integer> liste = Arrays.asList(1, 2, 3);
IntStream stream = liste.stream().mapToInt(Integer::intValue);
stream.forEach(System.out::println);
```

Résultat

```
1
2
3
```

Java

Arrays.stream() vs Stream.of()

- La méthode `Arrays.stream()` fonctionne seulement avec les tableaux de type primitif `int[]`, `long[]`, et `double[]`.
- Sa valeur de retour est respectivement `IntStream`, `LongStream` et `DoubleStream`.
- La méthode `Stream.of()` retourne un Stream générique de type `T Stream`. Par conséquent, et contrairement à `Arrays.stream()`, elle accepte tous les types.

Java

Pour convertir `IntStream` **en** `Stream<Integer>`

```
IntStream stream = IntStream.of(1, 2, 3);
Stream<Integer> ints = stream.boxed();
ints.forEach(System.out::println);
```

Java

Pour convertir `IntStream` **en** `Stream<Integer>`

```
IntStream stream = IntStream.of(1, 2, 3);
Stream<Integer> ints = stream.boxed();
ints.forEach(System.out::println);
```


Résultat

```
1
2
3
```

Java

IntStream vs Stream<Integer>

- `IntStream` : stream de primitifs de type `int`.
- `Stream<Integer>` : stream d'objets de type `Integer`.

Java

Exemple avec la méthode `range()`

```
var stream = IntStream.range(1, 3);
stream.forEach(System.out::println);
```

Java

Exemple avec la méthode `range()`

```
var stream = IntStream.range(1, 3);
stream.forEach(System.out::println);
```

Résultat

```
1
2
```

Java

Exemple avec la méthode `rangeClosed()`

```
var stream = IntStream.rangeClosed(1, 3);
stream.forEach(System.out::println);
```

Java

Exemple avec la méthode `rangeClosed()`

```
var stream = IntStream.rangeClosed(1, 3);
stream.forEach(System.out::println);
```

Résultat

```
1
2
3
```

Java

On peut aussi utiliser la méthode `iterate(valeurInitiale, expressionLambda)`

```
var stream = IntStream.iterate(1, i -> i + 1).limit(3);
stream.forEach(System.out::println);
```

Java

On peut aussi utiliser la méthode `iterate(valeurInitiale, expressionLambda)`

```
var stream = IntStream.iterate(1, i -> i + 1).limit(3);
stream.forEach(System.out::println);
```

Résultat

```
1
2
3
```

Java

On peut aussi convertir `IntStream` en `DoubleStream`

```
IntStream stream = IntStream.of(1, 2, 3);
DoubleStream stream1 = stream.mapToDouble(n -> n * 3);
stream1.forEach(System.out::println);
```

Java

On peut aussi convertir `IntStream` en `DoubleStream`

```
IntStream stream = IntStream.of(1, 2, 3);
DoubleStream stream1 = stream.mapToDouble(n -> n * 3);
stream1.forEach(System.out::println);
```

Résultat

```
3
6
9
```

Java

Pour calculer la somme des valeurs d'un `IntStream`

```
IntStream stream = IntStream.range(0, 6);
System.out.println(stream.sum());
// affiche 15
```

Java

Pour calculer la somme des valeurs d'un `IntStream`

```
IntStream stream = IntStream.range(0, 6);
System.out.println(stream.sum());
// affiche 15
```

Et pour calculer le maximum

```
IntStream stream = IntStream.range(0, 6);
System.out.println(stream.max().orElse(-1));
// affiche 5
```